Identifiers should be meaningful. While **`i = 1`** is perfectly valid, **`index = 1`** is preferable. When an identifier comprises two or more words, use camelCase (**`rowIndex`**), CapWords (**`RowIndex`**), or underscores (**`row_index`**.) Don't mix them; pick a style and stick with it.

Python Keywords and Operators

There are more than thirty Python keywords, including things like **`if`**, **`else`**, **`while`**, and **`import`**. The two Boolean constants **`True`** and **`False`** are capitalized; the other keywords are not. Rather than memorizing a list of keywords, learn them as you need them.

Python arithmetic operators include the usual suspects, **`+`**, **`-`**, **`*`**, and **`/`** for addition, subtraction, multiplication, and division. You can also use

Python also has many compound data types, including arrays and lists, data types for date and time, and for enumeration. You will learn these as you need them.

Assigning a value to a variable for the first time *declares* the variable, that is, creates it.

```
answer = 'yes'  # The variable "answer" exists and has a value
```

Python is *dynamically typed*. That means the data type of a variable can change when a new value is assigned to it.

```
answer = 'yes'  # The variable "answer" is a string
answer = 42     # And now it's an integer
```

It is, in general, good practice not to change the type of a variable during the execution of a program. Python has a built-in type( ) that will return the type of a variable is passed to it.[2]

```
>>> answer = 'yes'
>>> type(answer)
<class 'str'>
>>> answer = 42
>>> type(answer)
<class 'int'>
```

Block structure and Indentation

In many languages, indentation is used for pretty-printing, that is to make the program more readable, but is ignored by the language processor. In Python, indentation is used to delineate syntactic structures called *blocks*, and so it has meaning to the Python language processor. It also has the effect of making the program more readable.

```
if i < 0:                   # The colon starts a block
    print "i is negative"   # This is within the block
else:                       # A new block at the same level
   print "i is nonnegative"
   if i < 10:               # An inner block; indented further
     print "i has one digit"
   else:
     print "i has multiple digits"
```

Indentation must be consistent; “ragged” indentation will cause an error, as will mixing tabs and spaces. Python’s specified best practice in PEP-8 is to use four spaces for each block level.[3]

Defining and Using Functions

Python has hundreds of functions, like the **`type( )`** function described earlier. You can write your own functions, too. You’d do that when the same bit of code would otherwise have to

[2] The Python interpreter uses >>> to prompt for input from the user.
[3] That’s right… four keystrokes when one ought to do. Happily, autopep8 will fix things up for you. Some editors and IDEs can also help with this.

appear in two or more places within your program. A function means the code appears in only one place and is called from several places as needed.[4]

A function is defined by the keyword def (for "define") followed by the function name, the parameters of the function enclosed in parentheses, and a colon. The colon begins a block, and the function body – the code and data that comprise the function – are the contents of the block. *Parameters* are placeholders for data; when the function is called, the parameters are replaced by the *arguments* with which the function was called. If there are no parameters, the parentheses are empty but must still be present.

Here is a tiny Python program with an example of defining and using a function.

```
def greet(who, when):
    print("Good " + when + ", " + who + "!")

greet("Bob","afternoon")   # Prints "Good afternoon, Bob!"
player = "Gina"
time = "evening"
greet(player,time)         # Prints "Good evening, Gina!"
```

In this program, the function is called **`greet`** and the parameters are **`who`** and **`when`**. There's something a little strange, too. The print function is printing strings, but there are plus signs in there. The reason is that the plus operator is *overloaded* in Python to serve as the string concatenation operator, too. If plus is given two numbers, it adds; if it's given two strings, it concatenates. Mixing strings and numbers gives an err2 ((a)4)3 (o f)3!P03 Tc 0.0030D-o5o-345 >>BDC -0. a #

To use an object, it must be defined either within its own scope or within an enclosing scope. "Scope" only means whether an object is visible to a particular piece of code. Let's look at an example:

```
# Program to illustrate "scope"
name = "Gina"

def greetTest():
    print ("Entering greetTest")
    print (name)        # Prints "Gina"
    when = "morning"
    print (when)        # Prints "morning"
    print ("Leaving greetTest")

greetTest()
print (name)    # Works as expected, prints "Gina"
print (when)    # Name Error: name 'when' is not defined
# End of the program
```

There are two important things to notice about this example. The variable **`name`** is defined in the outer scope, and so is available anywhere in the program, including within the function **`greetTest`**. The variable **`when`** is defined within **`greetTest`**; it is available within the scope of **`greetTest`**, including any inner scopes – there aren't any inner scopes in the example – but it is not available outside the scope of **`greetTest`**.

If the same name is defined in an outer and inner scope, the definition in the current scope is used. Such a definition makes the name a *local variable*. Let's look at the ex (s)-132.14 -1.38 Td(out)-va l

```
name = "Gina"
def greetTest():
    global name          # 'name' will NOT be redefined
    name = "George"      # The variable OUTSIDE the function changes
    print (name)
greetTest()
```

We no longer have to qualify **`sleep`** by prefixing the module name, so we type less. Cool! The whole time module is still imported, but only the sleep() method becomes a part of the current program's name space.

Instantiating and Using Objects

The Error Messages Are Your Friends

One of the sad things about learning to program is that we make mistakes. If we're lucky, the mistakes lead to error messages.[8] Since we don't like making mistakes, those error messages sometimes make our brains slam shut, and that's a bad thing.

So, *read* the error messages; they exist to help you. Python's error messages will tell you where (at what line) the error was *detected*, how Python got there, and something about the error. Note that where the error was detected isn't always where the error is. You may have to put in some effort! For example, suppose a message says, "Name Error: name 'when' is not defined." The actual error is almost certainly somewhere else. Perhaps **`when`** wasn't defined at all, or it was misspelled, perhaps as **`wjen`**, when it was defined. On the other hand, if the message says, "Name Error: name 'wjen' is not defined" then the error is probably right there, in the form of a misspelled variable name.

A Sample Program

Here is a Python program taken from https://www.programiz.com/python-programming/examples/prime-number Read it carefully. Identify those features you learned here and look up anything that's new. That's how you learn to read and write!

```
# Python program to check if the input number is prime or not
# take input from the user
num = int(input("Enter a number: "))  # Look up int and input
# prime numbers are greater than 1
if num > 1:
   # check for factors
   for i in range(2,num):             # Look up for…in
       if (num % i) == 0:
# Look back at Keywords and Operators; look up modulus, maybe
           print(num,"is not a prime number")
           print(i,"times",num//i,"is",num)
           break
   else:
       print(num,"is a prime number")

# if input number is less than
# or equal to 1, it is not prime
else:
   print(num,"is not a prime number")
```

[8] If we're unlucky, our programs run, but give wrong answers with no indication of error. Language processors like Python can detect *syntax errors*, that is, errors in the expression of the language. They usually can't detect *logic errors*, in which we do the wrong thing, but in the right way.